Programming Languages and Techniques (CIS120)

Lecture 1

January 15, 2014

Course Overview and Logistics
Introduction to Program Design

Welcome!

Introductions

- Instructor: Dr. Stephanie Weirich
  - sweirich@cis.upenn.edu
  - Levine 510
  - http://www.cis.upenn.edu/~sweirich
  - Office hours: Mondays 1-3 PM or by appointment

- Course Administrator: Laura Fox
  - Levine 308
  - lffox@cis.upenn.edu

TA Staff*

- Harmony Li
- Tiernan Garsys
- Doron Shapiro
- Jinesh Desai
- Xander Goldman
- Jason Kong
- Ian Sibner
- Sudarshan Muralidhar
- Raul Martinez
- Xiuruo Zhang
- Jordan Hurwitz
- Thomas Delacour
- Cliff Kao
- Pearl Li
- Nancy Wong
- CJ Cobb
- Samy Lanka
- Lukáš Vacek
- Jack Gindi
- Meyer Kizner

*AKA: CIS 120 spirit guides, student champions, and all-around defenders of the universe.

What is CIS 120?

- CIS 120 is a course in **program design**
- Practical skills:
  - ability to write larger (~1000 lines) programs
  - increased independence ("working without a recipe")
  - test-driven development, principled debugging
- Conceptual foundations:
  - common data structures and algorithms
  - several different programming idioms
  - focus on modularity and compositionality
  - derived from first principles throughout
- It will be fun!

Prerequisites

- We assume you can already write 10 to 100-line programs in some imperative or OO language
  - Java experience is *strongly recommended*
  - CIS 110 or AP CS is typical
  - You should be familiar with using a compiler, editing code, and running programs you have created

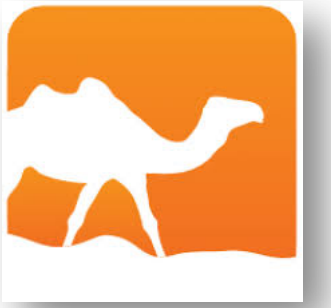
- CIS 110 is an alternative this course
  - If you have doubts, come talk to me or one of the TAs to figure out the right course for you

Philosophy

- Teaching introductory computer science
  - Start with basic skills of "algorithmic thinking" (AP/110)
  - Develop systematic design and analysis skills in the context of larger and more challenging problems (120)
  - Practice with industrial-strength tools and design processes (120, 121, and beyond)
- Role of CIS120 and program design
  - Start with foundations of programming using the rich grammar and precise semantics of the OCaml language
  - Transition (back) to Java *after* setting up the context needed to understand why Java and OO programming are good tools
  - Give a taste of the breadth and depth of CS

CIS 120 Tools

- OCaml
  - Industrial-strength, statically-typed Functional Programming language
  - Lightweight, approachable setting for learning about program design



- Java
  - Industrial-strength, statically-typed Object Oriented language
  - Many tools/libraries/resources available

- Eclipse
  - Popular open-source integrated development environment (IDE)

Installation: http://www.seas.upenn.edu/~cis120/current/ocaml_setup.shtml

Why *two* languages?

- Pedagogic progression
- Disparity of background
- Confidence in learning new tools
- Perspective

"[The OCaml part of the class] was very essential to getting fundamental ideas of comp sci across. Without the second language it is easy to fall into routine and syntax lock where you don't really understand the bigger picture.''
---Anonymous CIS 120 Student

"[OCaml] made me better understand features of Java that seemed innate to programming, which were merely abstractions and assumptions that Java made. It made me a better Java programmer.''
--- Anonymous CIS 120 Student

Administrative Matters

http://www.seas.upenn.edu/~cis120/

Registration

- If you are not currently registered, add your name to the *wait list* linked from the course web page

- If you need to switch recitations and the recitation you would like is closed, fill out the online *change request form* linked from the course web page

- Recitations start next week: Go to first meeting of the recitation section you want to join
- New recitation, Weds 1-2 PM!

Course Components

- Lectures (2% of final grade)
  - Presentation of ideas and concepts
  - Interactive demos
  - Grade based on participation using "clickers"
  - Lecture notes available on course website. Read Chapter 1!
- Recitations (6% of final grade)
  - Practice and discussion in small group setting
  - Grade based on participation
- Homeworks (50% of final grade)
  - Practice, experience with tools
  - Exposure to broad ideas of computer science
  - Grade based on automated tests + style
- Exams (42% of final grade)
  - In class, pencil and paper
  - Do you understand the terminology? Can you reason about programs? Can you synthesize solutions?

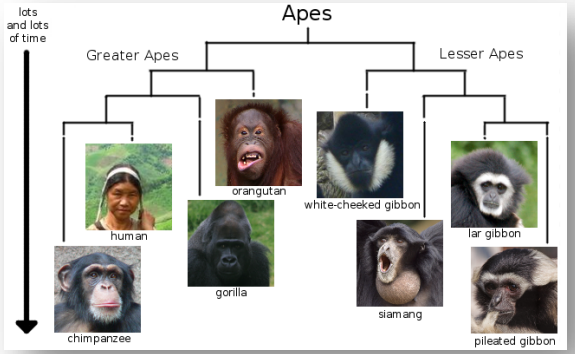
Clickers

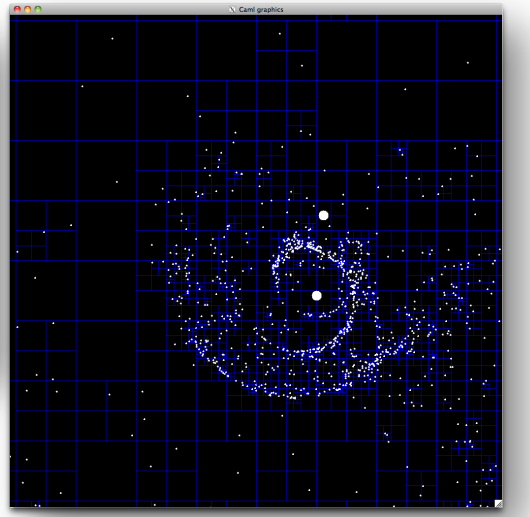
- We will use TurningPoint ResponseCards (clickers) for interactive exercises during lectures.
  - wrong answers will not count against your grade
- Please buy one at the bookstore (textbook section)
  - You can sell it back at the end of the semester
- Bring it to lecture every day, beginning Friday
  - Participation grades start 1/27

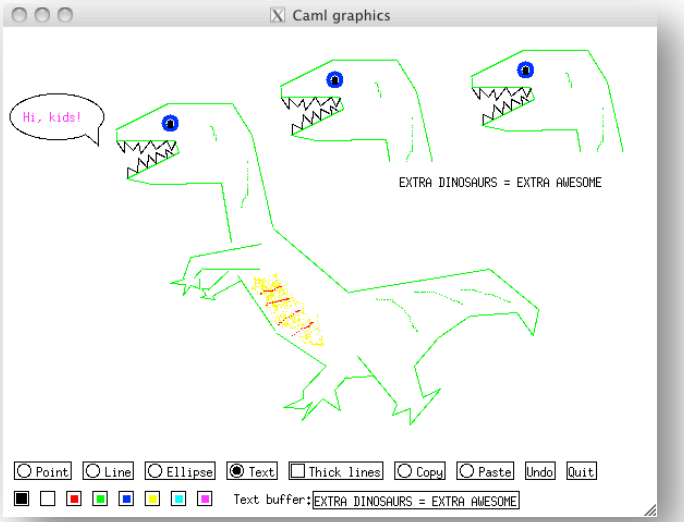
Lecture Policy

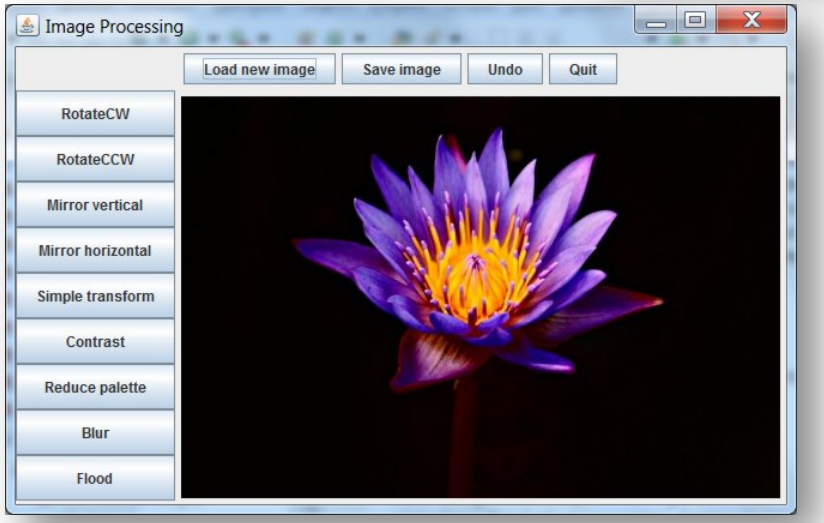
- Laptops *closed*… minds *open*
  - Although this is a computer science class, the use of electronic devices -- laptops, cell phones, mobile devices, iPads, etc., in lecture is prohibited.
- Why?
  - Laptop users tend to surf/chat/e-mail/game/etc.
  - They also distract those around them
  - You will get plenty of time in front of your computers while working on the course projects :-)

Some of the homework assignments…



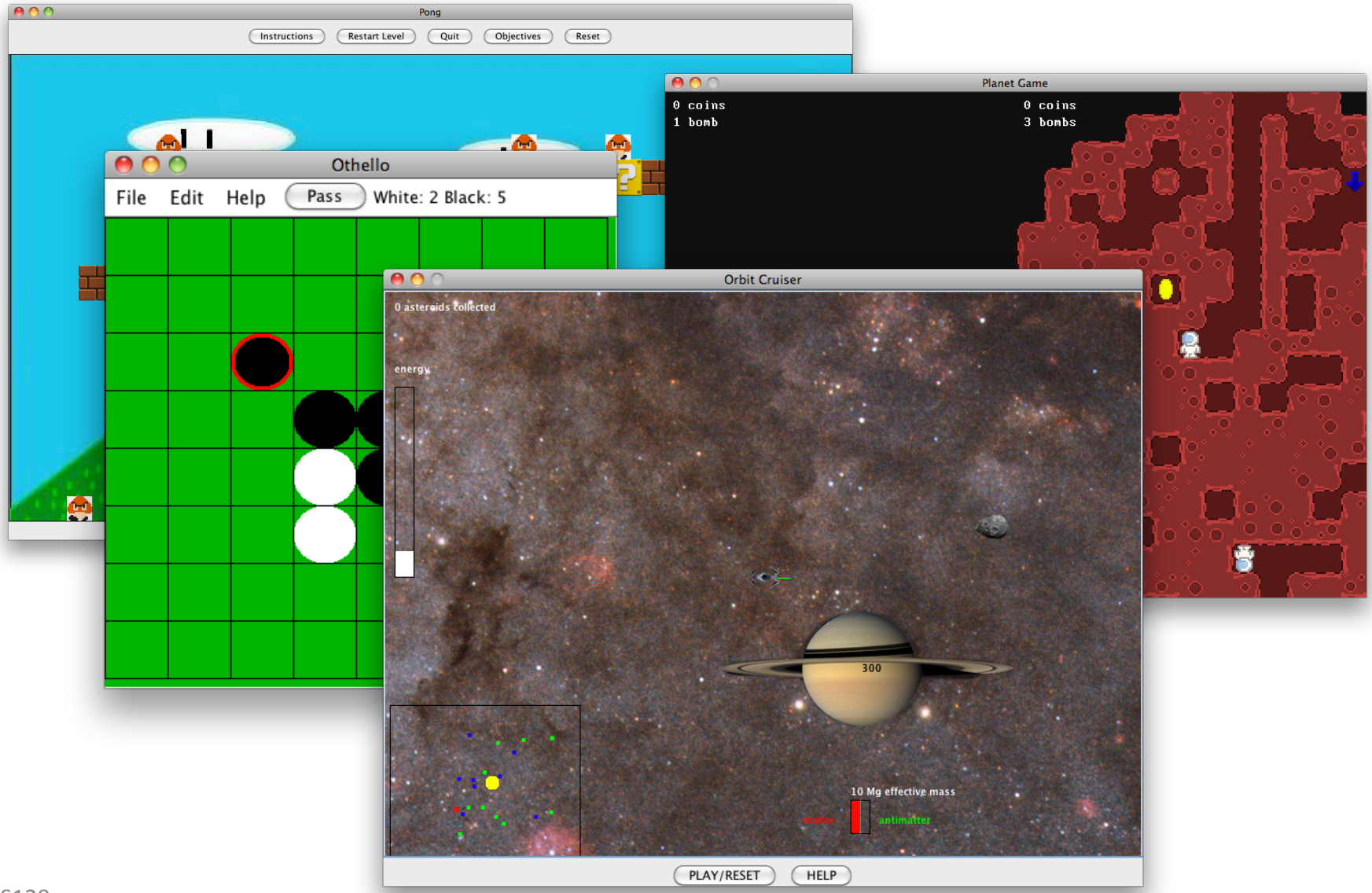






10 homework assignments total, weighted equally

Final projects



Academic Integrity

- Submitted homework must be *your individual work*

Not OK:
- Copying someone else's code

OK / encouraged:
- "High level" discussions of concepts

- Course staff will check for copying.

Violations will be treated seriously!

- *If in doubt, ask.*

Penn's code of academic integrity:
http://www.vpul.upenn.edu/osl/acadint.html

Academic Integrity

- Not OK

  A: I still can't figure out this problem on HW06. How do you write checkboxes?

  B: Oh, I'm done already. Yea, that problem took forever.

  A: Wait, you're done? Can I look at your code?

  B: Sure (*shows code*)

- OK

  A: I still can't figure out this problem on HW06. How do you write checkboxes?

  B: Oh, I'm done already. Yea, that problem took forever.

  A: Wait, you're done? Can I look at your code?

  B: Well, what are you stuck on?

  A: (*points to own code*) I don't get this thing about listeners...

  B: Oh! Those things are weird. So think of it this way...

  A: Ok, I understand now. Thanks!

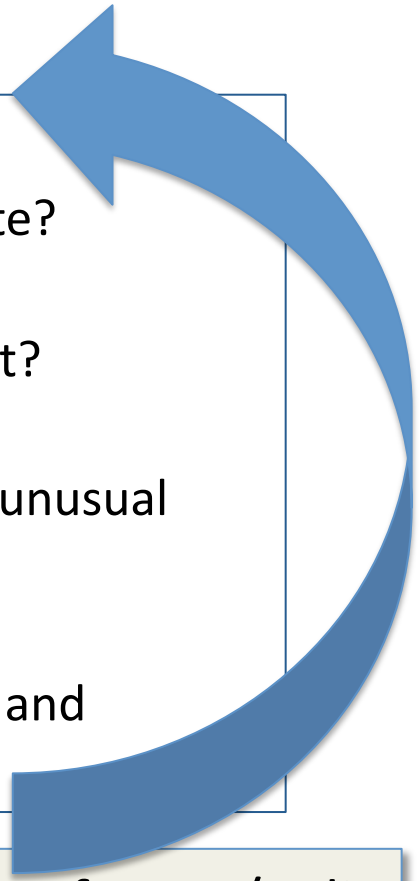
- Bottomline: your homework should come from *your* brain, as well as your fingers.

Program Design

Fundamental Design Process

Design is the process of translating informal specifications ("word problems") into running code.

1. Understand the problem
   What are the relevant concepts and how do they relate?
2. Formalize the interface
   How should the program interact with its environment?
3. Write test cases
   How does the program behave on typical inputs? On unusual ones? On erroneous ones?
4. Implement the required behavior
   Often by decomposing the problem into simpler ones and applying the same recipe to each



5. Revise / Refactor / Edit

A design problem

Imagine the owner of a movie theater who has complete freedom in setting ticket prices. The more he charges, the fewer people can afford tickets. In a recent experiment the owner determined a relationship between the price of a ticket and average attendance. At a price of $5.00 per ticket, 120 people attend a performance. Decreasing the price by a dime ($.10) increases attendance by 15. Unfortunately, the increased attendance also comes at an increased cost. Every performance costs the owner $180. Each attendee costs another four cents ($0.04). The owner would like to know the exact relationship between profit and ticket price so that he can determine the price at which he can make the highest profit.

Step 1: Understand the problem

- What are the relevant concepts?
  - *(ticket) price*
  - *attendees*
  - *revenue*
  - *cost*
  - *profit*
- What are the relationships among them?
  - profit = revenue – cost
  - revenue = price * attendees
  - cost = $180 + attendees * $0.04
  - attendees = *some function of the ticket price*

So profit, revenue, and cost also depend on price.

- Goal is to determine profit, given the ticket price

Step 2: Formalize the Interface

*Idea: we'll represent money in cents, using integers**

comment documents the design decision

type annotations declare the input and output types**

```
(* Money is represented in cents. *)
let profit (price : int) : int = …
```

* Floating point is generally a *bad* choice for representing money: bankers use different rounding conventions than the IEEE floating point standard, and floating point arithmetic isn't as exact as you might like. Try calculating 0.1 + 0.1 + 0.1 sometime in your favorite programming language…

**OCaml will let you omit these type annotations, but including them is *mandatory* for CIS120. Using type annotations is good documentation; they also improve the error messages you get from the compiler. When you get a type error message from the compiler, the first thing you should do is check that your type annotations are there and that they are what you expect.

Step 3: Write test cases

- By looking at the design problem, we can calculate specific test cases

```
let profit_500 : int =
   let price      = 500 in
   let attendees = 120 in
   let revenue    = price * attendees in
   let cost       = 18000 + 4 * attendees in
   revenue - cost
```

Writing the Test Cases in OCaml

- Record the test cases as assertions in the program:
  - the *command* run_test executes a test

a *test* is just a function that takes no input and returns true if the test succeeds

```
let test () : bool =
  (profit 500) = profit_500

;; run_test "profit at $5.00" test
```

the string identifies
the test in printed output
(if it fails)

note the use of double semicolons
before commands

Step 4: Implement the Behavior

Profit is easy to define:

```
let attendees (price : int) = ...

let profit (price : int) =
  let revenue = price * (attendees price) in
  let cost = 18000 + 4 * (attendees price) in
  revenue - cost
```

Apply the Design Pattern Recursively

attendees* requires a bit of thought:

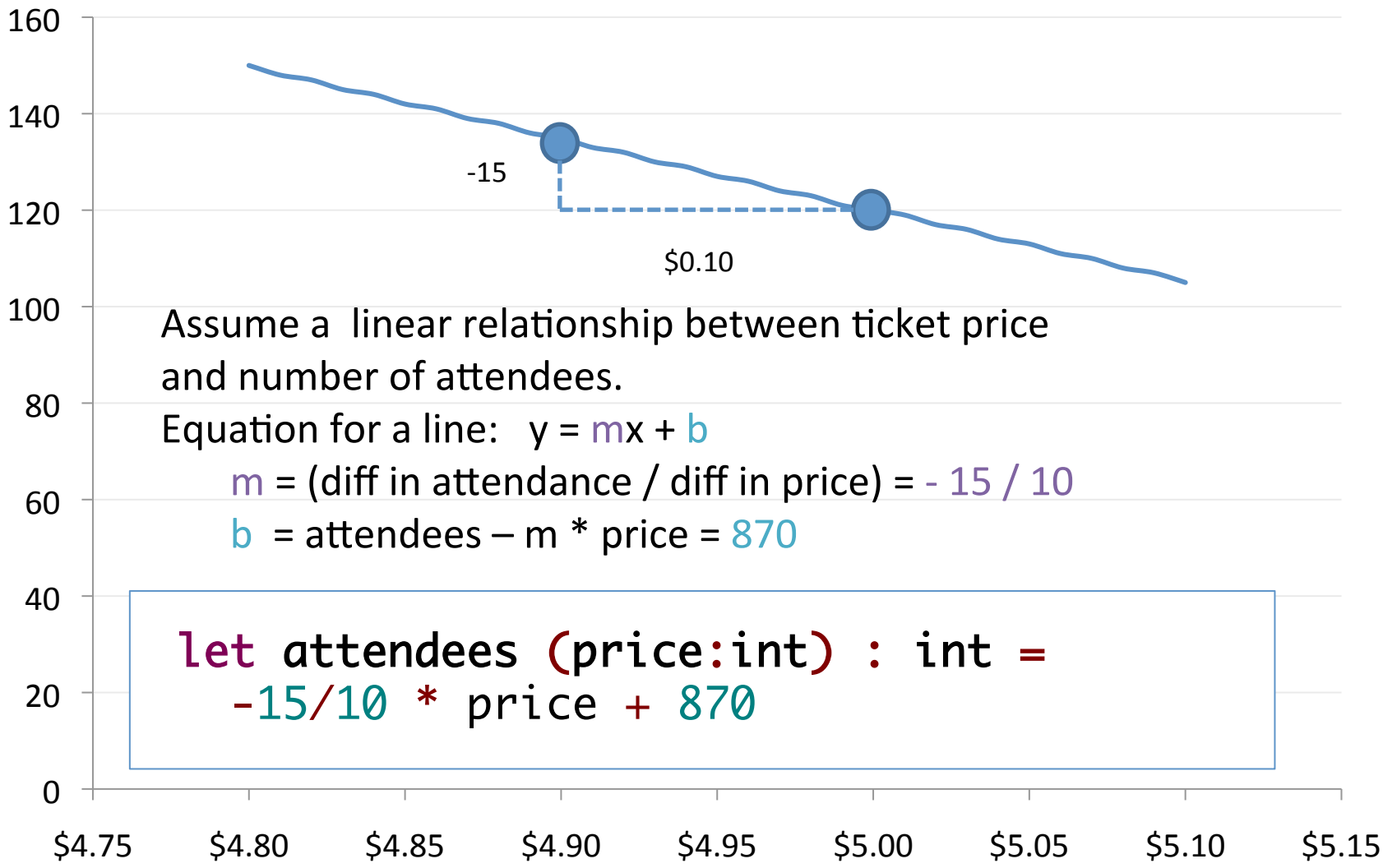
```
let attendees (price : int) : int =
   failwith "unimplemented"

let test () : bool =
   (attendees 500) = 120
;; run_test "attendees at $5.00" test

let test () : bool =
   (attendees 490) = 135
;; run_test "attendees at $4.90" test
```

"stub out" unimplemented functions

generate the tests from the problem statement *first*.

*Note that the definition of attendees must go *before* the definition of profit because it makes use of the attendees function.

Attendees vs. Ticket Price



Assume a linear relationship between ticket price and number of attendees.

Equation for a line: y = mx + b

m = (diff in attendance / diff in price) = - 15 / 10

b = attendees – m * price = 870

```
let attendees (price:int) : int =
  -15/10 * price + 870
```

Run!

Run the program!

- One of our test cases for attendees failed…
- Debugging reveals that integer division is tricky*
- Here is the fixed version:

```
let attendees (price:int) :int =
    (-15 * price) / 10 + 870
```

*Using integer arithmetic, -15 / 10 evaluates to -1, since -1.5 rounds to -1. Multiplying -15 * price before dividing by 10 increases the precision because rounding errors don't creep in.

Using Tests

Modern approaches to software engineering advocate *test-driven development*, where tests are written very early in the programming process and used to drive the rest of the process.

We are big believers in this philosophy, and we'll be using it throughout the course.

In the homework template, we may provide one or more tests for each of the problems. They will often not be sufficient. You should *start* each problem by making up *more* tests.

How *not* to Solve this Problem

```
let profit price =
    price * (-15 * price / 10 + 870) -
    (18000 + 4 * (-15 * price / 10 + 870))
```

This program is bad because it

- hides the structure and abstractions of the problem
- duplicates code that could be shared
- doesn't document the interface via types and comments

Note that this program still *passes all the tests!*

Summary

- *To read:* Chapter 1 of the lecture notes and course syllabus. Both available on the course website

- *To buy:* Turning Point clicker. Bring to every class, and register your ID number on the course website

- *To do:* Try to install OCaml and Eclipse on your laptops, following the setup instructions on the course website. TAs will hold office hours this week to help.